Parallel Sorting Algorithms

Three sorting algorithms (Quicksort, Bubblesort, Mergesort) are described below, which should be implemented sequentially first and then in parallel using OpenMP. Each team member should work on one of the algorithms.

All algorithms should be implemented in a .c program in such a way that a simple array of integer values is completely sorted. The size of the array should be specified using a "#define" macro. The array is then filled with random numbers and is to be sorted using the various algorithms.

If you have different ideas for parallelizing the algorithms (e.g. sections, tasks, for-directive), you are welcome to implement several versions of an algorithm.

a) Please develop a serial .c program for each algorithm.
b) Think about how you can parallelize the algorithms with OpenMP and try out different variants.
c) Compare the runtimes of serial and parallel programs using different sized arrays and different numbers of threads.

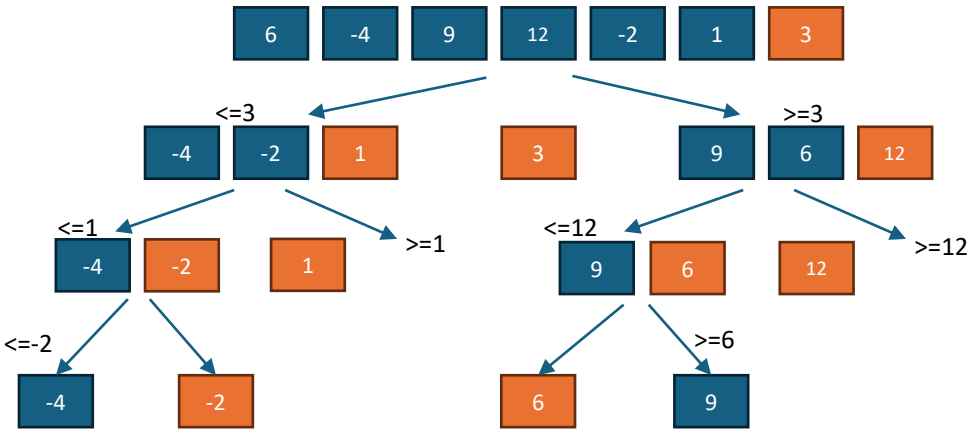
Please note: Solutions that are completely copied from external sources (books, internet, etc.) and are not referenced accordingly will be awarded 0 points.

Explanation-Paper (1-2 Pages per Group):
Compare the runtimes of the algorithms both serially and in parallel with a different number of threads. Also use different sized arrays for sorting. Decide on a system on which all tests are carried out under the same conditions (e.g. Linux Lab or Cluster) and specify this.

Present your results graphically (e.g. tables, graphs, ...), discuss the results, e.g. using terms such as “speedup”, “scalability”, “memory efficiency”, “overhead” ...

Important: Mark and describe in the text who has taken on which task and written which section.

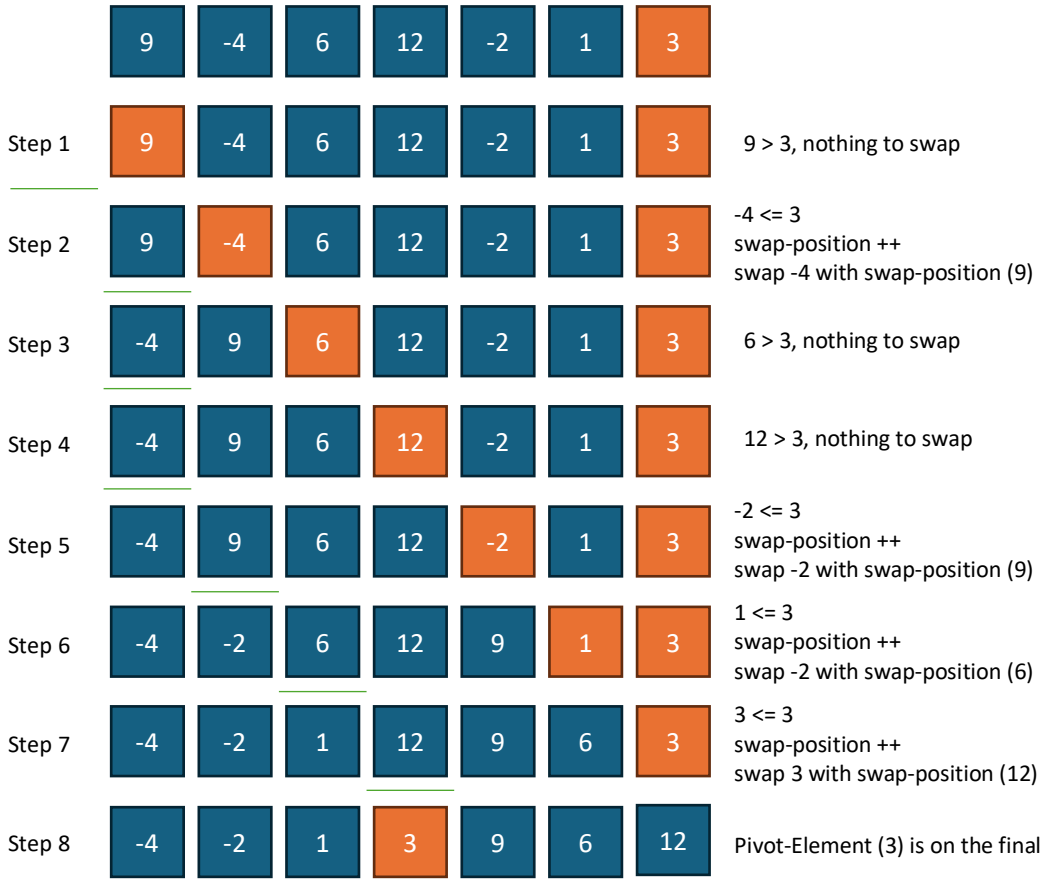
Quicksort



Quicksort is a sorting algorithm that is based on the divide-and-conquer strategy. An array of size n is to be sorted according to this algorithm as follows:

1. Determine pivot element: One element of the array forms the pivot element. The last element of the array can be used for this by default.

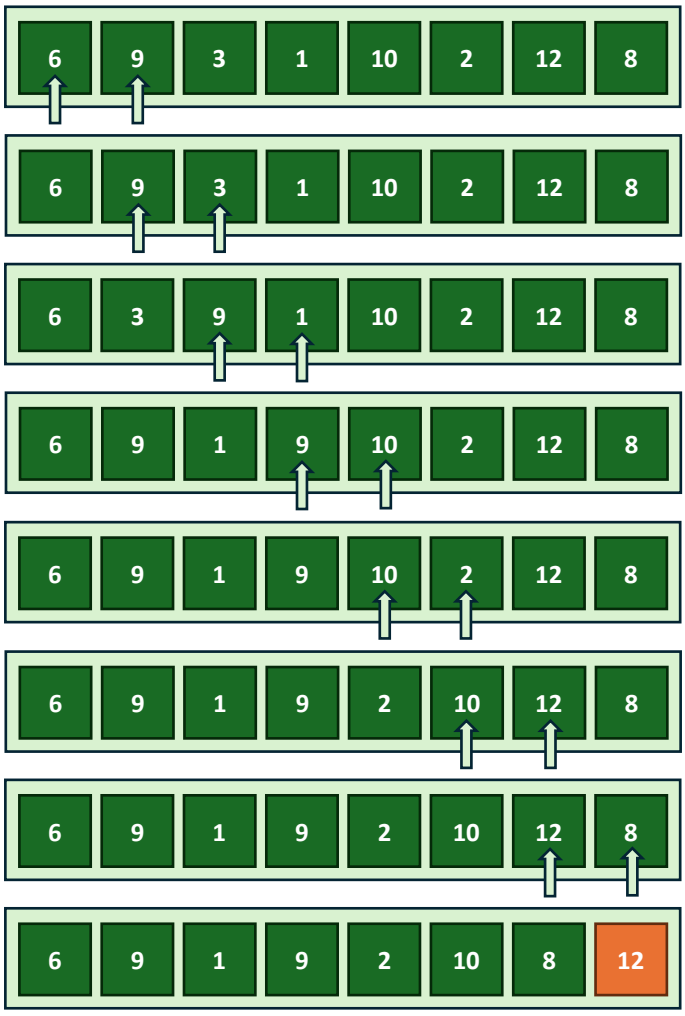
2. Partitioning: All values in the array that are smaller than the pivot element are arranged to the left of it (before it). All values in the array that are larger than the pivot element are arranged to the right of it (after it). After this step, the pivot element is at its final position.



3. Repeat the principle with the two resulting sub-arrays (one consists of the values smaller than the pivot element and one of the values larger than the pivot element). The principle is repeated until there are only sub-arrays consisting of one or no elements.

Bubblesort

In Bubblesort, we start on the left-hand side of the array and compare two neighboring numbers with each other. If the number on the left is greater than the number on the right, they are swapped.



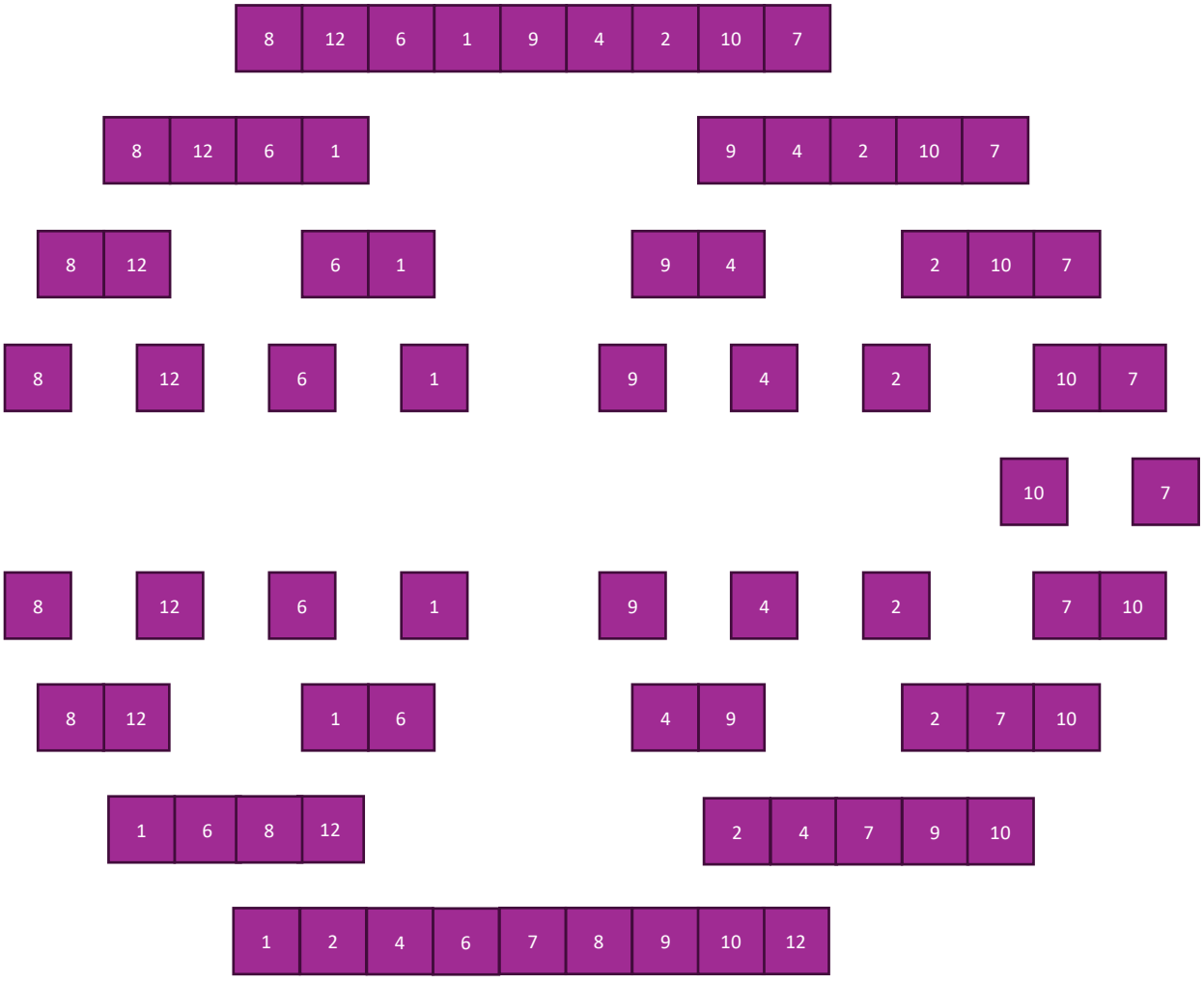
After the "sequence" shown, it is certain that the largest element is at the end of the array. The procedure is repeated until no more swaps occur during a run. The last value in its final position no longer needs to be included in the procedure for the next round.

Mergesort

Like the quicksort, the mergesort is based on the divide and conquer approach.

1. Dividing: Divide the array and the resulting subarrays into two halves again and again until it can no more be divided.

2. Merge and sort: In the opposite direction, two values/subarrays are now merged on each stage to create a sorted array.



Merge concept between two (sub)arrays:

On both arrays lies a merge index, which is initially located on the first element of the respective array. The two values on which the merge index is located are compared with each other. The smaller of the two is appended to the new array. The merge index of the array from which the value was taken is moved forward by one position. It remains unchanged in the other array. These steps are repeated until both arrays have been completely merged into one.